PASA: A SOFTWARE ARCHITECTURE FOR BUILDING POWER AWARE EMBEDDED SYSTEMS

Cristiano Pereira and Rajesh Gupta

Depart. of Information and Computer Science
University of California, Irvine

Mani Srivastava

Depart. of Electrical Engineering
University of California, Los Angeles

ABSTRACT

We present here a software architecture and an API that enables application developer to explore and build power / performance tradeoffs in the application software and enable the operating system to make appropriate hardware adjustments guided by application and system dynamic power management policies. We demonstrate the effectiveness of our implementation by implementing power aware scheduling schemes within eCos embedded real time operating system using real tasksets. We also present a battery driven adaptation scheme implemented in a wavelet based image compression application. The resuls show the usefulness of PASA (Power Aware Software Architecture) in power / performance tradeoffs and in comparison of power management policies in a real system implementation.

1. INTRODUCTION

Fueled by rapid advances in system integration and wireless communication technology, embedded systems are increasingly becoming networked. These systems often involve integration of high-performance computing and wireless communication capabilities, many of which operate under real-time constraints. The complexity of these systems together with the need for high- flexibility, and aggressive time-to-market schedules have resulted in the wide use of programmable system solutions. Software support for these systems usually takes the form of a real-time operating system (RTOS), device drivers, and runtime libraries.

Wireless embedded systems are often battery driven and, as a consequence, have to be designed and operated in a highly energy-efficient manner to maximize the battery lifetime. The pressing need for reduced-energy solutions has spurred the research and development of several low-power circuit design methodologies [3, 16]. While using low-power hardware circuits is necessary, it alone is not sufficient, especially since the increasing levels of system integration continuously worsen the energy problem. System lifetime can only be maximized by managing the various system resources in a power-aware manner, thus empowering the system with the ability to dynamically adjust its operating point in the performance-energy-fidelity tradeoff space. To address this issue, Dynamic Power Management (DPM) techniques have been investigated (see for example, [2]). A commonly used DPM scheme is to put the idle system components in a shutdown or into a low-power state. An alternative – and more efficient when applicable – technique is Dynamic Voltage Scaling (DVS), where the voltage and operating frequency of the processor are changed dynamically during runtime to just meet the performance requirement.

In this context, an RTOS provides a number of *services* to an embedded system application. It manages the creation, destruction, and scheduling of tasks, as well as the communication between tasks. It is responsible for all resource allocation and management decisions, and serves as an interface between an application and the underlying hardware platform. The RTOS has global information about the performance requirements and workload of all the applications, and can directly control the underlying hardware, tuning it to meet specific system requirements. These characteristics make the RTOS an ideal place to implement system-level power management policies. However, effective DPM cannot be done by the OS alone. It requires participation by the application software in ensuring that right power-performance tradeoffs are made in the context of the application. For this, we need a software interface that facilitates a continuous dialogue between the OS and the application. This dialogue enables the OS to serve as a broker between application-level functionality and parameter choices and the hardware capabilities for changing performance and power consumption.

1.1. Paper Overview and Contributions

This paper presents an structured and layered software architecture for power-aware wireless and portable embedded systems. The architecture consists of an RTOS kernel, and a set of standard software interfaces that enable easy exchange of timing and power information between the underlying hardware platform, the RTOS, and the application. It provides a programming interface (named PASA for Power-Aware Software Architecture) that can be used to efficiently incorporate system power management policies into the RTOS. To demonstrate the impact of PASA, we focus our attention on the task scheduling process in an RTOS. We use PASA to incorporate (and compare) different DPM / DVS techniques (from a simple shutdown based scheme to an advanced DVS scheme that enables adaptive power-fidelity tradeoffs) into the RedHat *eCos* [21] operating system running on a complete variable-voltage system based on the Intel XScale micro-architecture. We also collected power measurement results that show the efficiency of the different schemes implemented. It is important to note that this paper is not proposing nor presenting any innovative algorithm for power management at the OS level. Our chief contribution is a software and hardware platform that provides the capability to make power-performance tradeoffs and to assess effectiveness of DPM.

2. RELATED WORK

Many papers devising DVS techniques have been published in the last few years. Few of them, however, have presented real imple-

mentation of the algorithms on a real embedded systems platform running on a RTOS. Pillai and Shin [20] presented different algorithms for voltage/frequency scaling as well as simulation results and also the results on a implementation platform. The techniques were developed with embedded systems in mind but the implementation was realized on a laptop computer running Linux operating system, which is not a typical embedded system platform. The operating system kernel was modified by means of loadable modules in order to accomplish the implementation of the DVS techniques. The results and implementation details are presented in the referenced paper.

Flinn and Satyanarayana [6] developed a platform for energy-driven application adaptation in mobile devices. They showed that a constant and collaborative dialogue between operating systems and applications enables the adaptation of the latter based on the energy available and the battery life time duration required. By monitoring the energy supply and demand their platform is able to select the correct tradeoff between energy conservation and application quality. The techniques used are mostly based on lowering the fidelity of the application in order to save energy. Hardware based techniques such as turning off the display, hard disk and waveLAN card were also used. All the experiments were executed running Linux operating system on Pentium based laptops. No DVS schemes were implemented.

The importance of a well-structured software architecture for energy-efficiency in mobile systems is discussed in [24]. As a first step to analyze and improve the energy impact of various OS decisions, researchers have attempted to characterize the power consumption of embedded RTOSs [4, 1]. There exists a multitude of work on OS-directed dynamic power management. Shutdown based power management schemes [2] attempt to optimize the system's transition policy between several states, each of which is characterized by a performance and power consumption level.

An early work on software architecture to enable power management at OS level include BIOS-based Advanced Power Management (APM [11]). The principal limitation of APM is that the OS has no knowledge of the APM actions. Its follow on, the Advanced Configuration and Power Management (ACPI [12]) allows OS- directed power management by defining hardware registers and BIOS interfaces (table, control methods). ACPI primarily enables use of Intel-specific hardware mechanisms for power reduction. It does not provide any support for real time contraints. More importantly, it does not provide any mechanisms for the application software that allow it to use the operating system power reduction services or allow the OS to take advantadge of the application knowledge.

Variable voltage schemes for energy-efficient task scheduling have mostly targetted either workstation-like environments where latency is not an issue, and average throughput is the metric of performance [25, 7], or hard real-time systems, where a single timing violation may be catastrophic to system functionality [13, 9, 22, 14]. It is only recently that a new class of power-aware scheduling schemes has emerged that targets *soft-real time systems*, and permits a few deadlines to be missed [15, **?**], thereby adding another degree of freedom, namely *system fidelity* or *quality of computation*, to the design and operation of these systems. This is particularly important for wireless systems where missed deadlines and packet loss over the wireless link can be treated in a similar fashion providing a level of flexibility to enable aggressive power management.

Finally from the experimentation and implementation point of view, Farkas et al [5] presented energy/power measurements on the Itsy pocket PC platform. They used a DAQ board to collect the data and characterized power consumption of the platform running different benchmarks. They characterized power consumption of the processor running at three different frequencies and two different voltages. This work presents power consumption data for different subsystems without addressing the issue of a software architecture or how hardware active frequency/voltage scaling takes place.

3. SOFTWARE ARCHITECTURE REQUIREMENTS

In seeking to develop an architecture for the system and application software we briefly examine the requirements that such an architecture must satisfy. There are, of course, requirements related to the real-time nature of many embedded applications, many of which are satisfied by a range of available RTOSs. For instance, the OS should be able to monitor the real time parameters of task instances (e.g., deadlines). Further, and not usually present by most of the RTOS's the OS must also be able to monitor the system workload and to predict future execution times based on previous ones. The OS should also be able to manage the available hardware knobs for speed-power trade-off, adjusting them according to the system workload. This includes setting processor frequency and voltage, and setting the processor into low power states by means of simple and structured interface.

For efficient system-level power management, it is important that an application is able to monitor and control power related hardware "knobs" (such as processor voltage and frequency) as well as control and take advantadge of power aware operating system services (such as task scheduling). There is a need for mechanisms in the system software that allow efficient communication of energy, performance and accuracy tradeoffs for a given application. The electrical "knobs" must be made available to the operating system to enable the OS system writer to introduce power and energy awareness into traditional OS services.

Making an OS or runtime system aware of the system power and energy constraints is not sufficient in providing guarantees on how an application will perform in a power/energy constrained environment. The operating system services must also be available to the application programmer so that application can make use of this information in determination of power/energy dependent functionality and performance characteristics. Specifically, facilities are needed for an application to create and instantiate a task, taking into consideration the task timing parameters (period, deadline and execution time). The application should also be able to inform the operating system about the start and end of its computation and also about the expected remaining time of a given task instance (helping the OS to get a picture of the system workload).

While many dynamic power management strategies are specific to the underlying hardware and software, the application requirements for functionality and performance delivered under energy constraints can be specified independent of the platform being used. Given the diversity of hardware and software platforms used in portable, embedded and/or real-time systems, it is critical that power, energy and timing information are communicated through well-defined interfaces to ensure application portability across platforms. More concretely, to make the power management related software layers level operating system independent, all system calls to the native operating system should be done by means of a portable operating system standard interface such as

POSIX.

4. PASA ARCHITECTURE

As mentioned earlier, We view the notion of power awareness in the application and OS as a capability that enables a continuous dialogue between the application, the OS, and the underlying hardware. This dialogue establishes the functionality and performance expectations (or even contracts, as in real-time sense) within the available energy constraints. We describe here our implementation of a specific service, namely the task scheduler, in PASA that makes it power aware. PASA is composed of two software layers and the RTOS kernel. One layer interfaces applications with operating system and the other layer makes power related hardware "knobs" available to the operating system. Both layers are connected by means of corresponding power aware operating system services as shown in Figure 1. At the topmost level, embedded applications call the API level interface functions to make use of a range of services that ultimately makes the application energy efficient in the context of its specific functionality. The API level is separated into two sub-layers. PA-API layer provides all the functions available to the applications, while the other layer provides access to operating system services and power aware modified operating system services (PA OS Services). Active entities that are not implemented within the RTOS kernel should also be implemented at this level (threads created with the sole purpose of assisting the power management of an operating system service, such as a thread responsible for killing threads whose deadlines were missed). We call this layer the power aware operating system layer (PA-OSL).

To interface the modified operating system level and the underlying hardware level, we define a power aware hardware abstraction layer (PA-HAL). The PA-HAL gives the access to the power related hardware "knobs" in a way that makes it independent of the hardware.

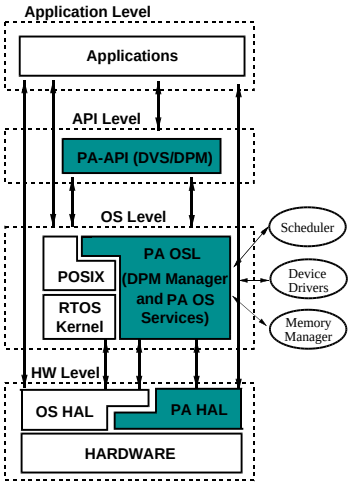

Figure 1. Power Aware Software Architecture

Table 1 lists the functions relevant to the implementation of power aware scheduling techniques. At the PA-API layer there are functions to create types (informing the real time related parameters) and instances of tasks, to notify start and end of tasks (needed by the OS in order to detect whether the task execution is over and the deadline of a task has been met), and to either inform the application about the execution time predicted by the OS or tell the OS about the execution time prediction estimated by the application (which can be based on application specific parameters). At the PA-OSL layer there are functions to manipulate information related to the power aware scheduling schemes that are maintained within the kernel (such as the type table in the case of the scheduler), the thread responsible for killing threads whose deadlines were missed (assuming that the threads whose deadlines were missed are no longer useful). The alarm handler notifies the killer thread, which in turn kills the thread and re-creates it. The overhead of having an extra thread is minimum since the killer thread is constantly blocked unless a request to kill another thread is received. When it happens the killer thread wakes up and finishes the execution of the proper thread. At the PA-HAL layer functions to manipulate processor frequency and voltage levels and low power states are present. These are called by the RTOS scheduler when slowing down the processor or shutting it down. For processor frequency and voltage scaling, different platforms have different precautions that have to be taken care of before doing the scaling. These precautions might have to be done before the scaling, after it or both before and after. For these the functions `pahal_dvs_pre_set_frequency_and_voltage` and `pahal_dvs_post_set_frequency_and_voltage` are provided and must be implemented by the OS programmer according to the platform. And finally functions to poll the status of battery based platforms are also important in order to enhance their lifetime.

Layer	Functions/Threads
PA-API	paapi_dvs_create_thread_type() paapi_dvs_create_thread_instance() paapi_dvs_app_started() paapi_dvs_get_time_prediction() paapi_dvs_set_time_prediction() paapi_dvs_app_done() paapi_dvs_set_adaptive_param() paapi_dvs_set_policy() paapi_dpm_register_device()
PA-OSL	paosl_dvs_create_task_type_entry() paosl_dvs_create_task_instance_entry() paosl_dvs_killer_thread() paosl_dvs_killer_thread_alarm_handler() paosl_dpm_register_device() paosl_dpm_deamon()
PA-HAL	pahal_dvs_initialize_processor_pm() pahal_dvs_get_frequency_levels_info() pahal_dvs_get_current_frequency() pahal_dvs_set_frequency_and_voltage() pahal_dvs_pre_set_frequency_and_voltage() pahal_dvs_post_set_frequency_and_voltage() pahal_dvs_get_lowpower_states_info() pahal_dvs_set_lowpower_state() pahal_dpm_device_check_activity() pahal_dpm_device_pre_switch_state() pahal_dpm_device_switch_state() pahal_dpm_device_post_switch_state() pahal_dpm_device_get_info() pahal_dpm_device_get_curr_state() pahal_battery_get_info()

Table 1. PASA relevant functions

Figure 2 shows an example on how the PA-API functions are used in a MPEG decoder source code when creating threads using PA-API functions. A thread is created specifying that the deadline

```
void main()
{
  mpeg_decoding_t =
    paapi_dvs_create_thread_type(100,30,100);

  paapi_dvs_create_thread_instance(
    mpeg_decoding_t, mpeg_decode_thread);
}
...
void mpeg_decode_thread()
{
  for (;;) {
    paapi_dvs_app_started();
    /* original code */
    mpeg_frame_decode()
    paapi_dvs_app_done();
  }
}
```

Figure 2. Power aware source code example for a MPEG decoder

and period are 100 and the worst case execution time is 30 (assuming it was profiled and therefore known ahead of time. The thread is instantiated and access to the power-aware functionality contracts is enabled and terminated by the functions `paapi_dvs_app_started()` and `paapi_dvs_app_done()` respectively. These functions delimit the work done by the threads which is encapsulated in one single function in this example.

There are also function meant to set the DVS scheme or the DVS scheme characteristics at the run-time. `paapi_dvs_set_policy()` sets the DVS policy to be used before the system initialization. `paapi_dvs_set_adaptive_param()` configures the parameters of the adaptive scheme that will be described in section 5.2. These functions, along with the function with the DPM prefix have not been fully implemented yet. `paapi_dpm_register_device()` register the device to be power managed. The function tells which device to be managed and which policy should be used to managed the device. Function `paosl_dpm_register_device()` access the device table within the kernel and updates it information. Function `paosl_dpm_deamon()` is responsible for monitoring idleness of a certain device and based on how long it has been idle switch it to low power modes. The policies on how to decide which state to switch to and after how long are out of the scope of this paper.

The function `pahal_dpm_device_check_activity()` tells for how long the device has been idle. The implementation of this function may rely on information taken from the device driver which provides access to the device. The functions `pahal_dpm_device_pre_switch_state()`, `pahal_dpm_device_switch_state()` and `pahal_dpm_device_post_switch_state()` are responsible for switching the device to the state passed in as parameter. The last two functions `pahal_dpm_device_get_info()` and `pahal_dpm_device_get_curr_state()` just provide information about the device and its current state.

5. PASA IMPLEMENTATION

The PASA software architecture presented above has been incorporated in the *eCos* operating system [1]. We ported *eCos* to an Intel XScale processor based platform called 80200 Intel Evaluation Board (80200 Board for short) [10].

The XScale platform supports nine frequency levels ranging from 200MHZ to 733Mhz, even though only seven of them are used in the 80200 board due to its own limitations. The processor can also be put on three different low-power modes: IDLE, DROWSY and SLEEP. The SLEEP state is the most power saving one but requires a processor reset in order to return it to active mode. The idle state, on the other hand, is the least power saving but requires a simple external interrupt to wake the processor up. We use only IDLE power down mode in our experiments due to its simple implementation.

As is the case with most RTOSs, *eCos* requires a periodic interrupt to keep track of the internal operating system tick, responsible for the timing notion within the system. In the 80200 board the only source of such interrupt is the internal XScale performance counter interrupt. In our case, this turned out to be a problem because the interrupt is internal to the processor. Therefore it cannot wake it up from one of the low power modes. Instead, we use a source of external interrupts to awaken the processor. The external interrupt is connected into the interrupt pin of the processor and the interrupts are generated from an external Altera FPGA board name UP1 (University Program 1). The relationship among the 80200 board, the FPGA board and the host-PC is depicted in Figure 3 (See a picture of the hardware setup in Figure 5). The Maxim board shown in the lower left part is a wire-wrapped board that is directly responsible for the dynamic voltage scaling of the XScale platform.

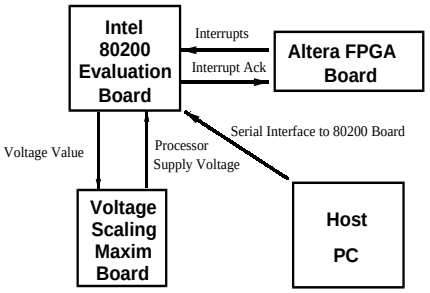

Figure 3. Hardware Architecture

5.1. Voltage Scaling

The hardware platform for implementing dynamic voltage scaling (voltage scaling at runtime) consists of a Maxim 1855 evaluation board, and an interface circuitry made using a PLD. Whenever a voltage change of the processor is required, the processor sends a byte through the peripheral bus of the 80200 board to interface circuitry which basically acts as an addressable latch. The outputs of this latch are connected to the digital inputs of the Maxim variable supply board. These inputs select the output supply voltage of the Maxim board and the processor analog and core supplies are fed from this supply voltage. For the experiments, the system was configured to run at supply voltages from 1.0V to 1.5V and frequencies from 333 MHz to 733 MHz according to the Table 2. The frequency is changed using XScale internal registers. The supply voltage from the Maxim board can vary at steps of 0.05V.

[1] The source code for PASA implementation can be downloaded from http://www.ics.uci.edu/∼cpereira/pads

Processor Frequency (Mhz)	Voltage Supply Level
733	1.5
666	1.4
600	1.3
533	1.25
466	1.2
400	1.1
333	1.0

Table 2. Frequencies and corresponding voltage levels for the Intel Xscale processor used in our implementation

In order to prove the utility of our software architecture, we implemented different power management algorithms using PASA. In the sequel we describe opportunities during task scheduling to both slowdown and shutdown the processor.

5.2. Power Aware Scheduling Algorithms

It has been observed in many systems that, during runtime, the processor utilization factor is often far lower than 100%, resulting in long idle intervals. This inherent slack can be exploited for DPM by either shutting down the processor, or through the use of processor slowdown and DVS. The extent of slowdown is limited by the schedulability of the task set at the reduced speed, since excessive slowdown may cause deadline violations. A second opportunity for DPM arises due to the time-varying nature of the system workload. Performance analysis studies have shown that for typical embedded system applications (*e.g.*, audio and video encoding/decoding, encryption/decryption *etc.*), the instance to instance task execution time varies significantly and is often far lower than the worst case execution time (WCET) [18], which results in additional slack being created in the task schedule. Since task instance execution times are not known at design time, to exploit this observation, the power management policy has to be dynamic in nature.

System shutdown. The most obvious way of exploiting idle intervals in the schedule is to shutdown the processor. Modern embedded processors offer multiple low-power states, corresponding to varying degrees of system shutdown. Most shutdown based DPM policies are ether predictive [23], or stochastic in nature. Several techniques for DPM policy optimization have also been proposed [2].

Static voltage and frequency scheduling. Quite often, in order to reduce the complexity of implementation, power management policies are static in nature. Note that *static* in the context of such a DVS policy refers to the fact that the voltage and frequency settings of the processor are determined offline at design time. The settings can vary from task to task, which will require the processor voltage and frequency to be changed dynamically during runtime. The main advantage of a static power management policy is that it is simpler than a dynamic policy. However, the downside is that such a policy cannot exploit DPM opportunities that arise dynamically, during system operation. We have implemented a static DVS technique described in [**?**]. The objective of the algorithm is to determine a slowdown factor for every task such that energy consumption is minimized while still guaranteeing the schedulability of the task set. The details on how to calculated the static slow down frequency factors are presented in details in Gruian's work [8].

The **response time** of a task instance is defined as the amount of time needed for the task instance to finish execution, from the instant at which it arrived in the system. The worst case response time (WCRT), as the name indicates, is the maximum possible response time that a task instance can have. For a conventional fixed speed system, the WCRT of a task under the RM scheduling scheme is given by smallest solution of the equation [19]:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \times C_j \quad (1)$$

where R is the response time, T the task period, C the task execution time and $hp(i)$ denotes all the tasks with priority higher than *i*.

The summation term on the right-hand-side of the equation represents the total interference that an instance of task *i* sees from higher priority tasks during its execution. If the WCET of the task is less than its deadline D_i, the task is guaranteed to be schedulable.

A greedy, iterative technique is used to solve this optimization problem. The scheme is enhanced with dynamic components, which are described next.

Proactive DVS scheme for power-fidelity tradeoff. In [**?**] the authors observe that even though task instance execution times vary significantly, they depend on data values that are obtained from physical real-world signals (*e.g.*, audio or video streams), and hence are likely to have some temporal correlation between them. This temporal correlation can be exploited to proactively manage processing resources by predicting the execution times of individual task instances based on the past history of execution times, and setting the processor speed and voltage accordingly. Several prediction models can be used, such as simple average, exponential average, least mean square, *etc*. While the use of more sophisticated models yields better prediction accuracy, it also places a higher computational burden on the scheduler, which is undesirable. As in [**?**], we use a simple average model that is computationally light-weight and reasonably effective.

Adaptive speed setting policy. As in any predictive policy, mispredictions do occur, which lead to task deadline misses. Wireless embedded systems invariably have some communication noise (in the form of data losses and packet errors), and are designed to be tolerant to these channel impairments. The result of a few task deadline misses is no different from noise, and only leads to a slight drop in *system fidelity*. To keep the system fidelity under specified limits, an adaptive feedback mechanism was introduced into the prediction process. The recent deadline miss history is monitored, and if the number of deadline misses is found to be increasing, the prediction is made more conservative, reducing the probability of further deadline misses. Similarly, a decreasing deadline miss history results in more aggressive prediction to reduce energy consumption. The prediction scheme thus becomes adaptive to a recent history of missed deadlines, resulting in an adaptive power-fidelity tradeoff which can be fine tuned to suit application needs.

Using this scheme, we keep slowing down the processor by a factor *S* when less than *T* deadlines are being missed in the last *W* threads executions. This process is repeated until the total slow down factor reaches a lower bound *M*. When *T* deadlines are missed in the previous window *W* of tasks executions, the processor is speeded up by factor *I*.

6. EXPERIMENTS AND RESULTS

We executed four different DVS algorithms using the software and hardware platforms previously described. The first is a simple

"shutdown when idle" scheme. The second applies static slowdown factors and shutdown. The third applies shutdown, static and dynamic slowdow factors, the latter based on average predictions. Finally the fourth scheme combines shutdown, static and dynamic slowdown factors, and in addition a deadline miss driven adaptive factor that works as described earlier. For the adaptive scheme we used $T = 2$, $W = 10$, $I = 0.1$ and $S = 0.05$. The values for M varied from 1.0 to 0.75. The analysis for selectin the best configuration of these numbers is out of the scope for this paper. Our goal here is to show the effectiveness of PASA in implementing and comparing different schemes configuration on a real system implementation.

For our prototype system, we used three different real applications running concurrently. The applications are an MPEG2 decoder, an ADPCM (Adaptive Differential Pulse Code Modulation) speech enconding and a floating point FFT algoritm. These applications are running concurrently. Each application is executed by an eCos thread under rate-monotonic scheduling. The periods are chosen carefully so that the task set is schedulable. We tried different schedulable periods for each task in order to demonstrate how it affects the power consumption on the system. For scheduling the task according to the RMS criteria, we had to profile the execution time of each application for different input data in order to collect the worst case execution times (WCET). For the MPEG2, different files were decompressed and for each one of them the WCET was measuread separately. We also implemented the FFT using a random gaussian distribution function that defines the number of FFT's computed by the FFT algorithm in each execution. Table 4 shows the characteristics of the applications used. The worst case execution time of each instance of the applications are measured when the processor is running at maximum frequency. The stardard deviation is also presented to show the variability on the execution times for each application.

All algorithms used in the experiments shutdown the processor as soon as it becomes idle. The processor is awakened when the next external interrupt arrives and then continues executing. The period of the fastest task is the same as the period of the external interrupt to ensure that at awaken time the processor is not idle. A limitation of the processor wake up strategy for our current testbed requires us to make all tasks with periods multiple of the fastest task. In this way whenever the processor is waken up there is usefull work to be done and so we do not wake it up for nothing. If we had a programmable interrupt generator this limitation would not exist anymore. However, this is not a limitation of the software architecture or the DPM algorithms.

Task	Application	WCET at Maximum Frequency (us)	Std Dev (us)
T1	MPEG2 (wg_gdo_1.mpg)	30700	3100
T2	MPEG2 (wg_cs_1.mpg)	26300	2100
T3	ADPCM	9300	3300
T4	FFT	15900	0
T5	FFT (gaussian dist.)	13600	800

Table 3. Applications used in the experiments

The shutdown mechanism is implemented using the operating system idle thread. Whenever the system becomes idle, the idle thread is scheduled. We take advantadge of this and run the code to switch the processor to IDLE state as the first action of this task. When the processor wakes up this task is resumed and the highest rate task starts executing.

For the algorithms described earlier, the slowdown factors are maintained internally to *eCos* in the form of tables. Each task type is associated with a static factor, which is computed during system initialization. The dynamic and adaptive factors are maintained in a table of task instances. The task type table also maintains a specified number (10 in our experiments) of execution times of previous tasks executions. *eCos* kernel has full access to these tables. PA-API layer accesses these tables by means of functions provided by the PA-OSL layer.

With all these information available, whenever a context switch occurs, the information about the preempted and scheduled tasks are updated and the voltage and frequency of the scheduled task are adjusted according to the static, dynamic and/or the adaptive factors, depending on to the algorithm running at the moment.

Table 4 shows the tasksets used during the experiments. The tasksets consist of combining the three applications mentioned earlier using different data files and periods. The static slowdown factor is also presented for each taskset. The WCET is obviously the same as the ones shown in Table 4. In the characteristics column for each task the Period, the WCET and the Deadline are presented respectively.

Taskset	Characteristics (us)	Static Slowdown Factor
A	T1 = (26300, 40000, 40000)	0.9495
	T3 = (9300, 80000, 80000)	0.9495
	T4 = (15900, 120000, 120000)	0.9495
B	T2 = (30700, 47000, 47000)	0.8979
	T3 = (9300, 94000, 94000)	0.8979
	T4 = (15900, 141000, 141000)	0.8979
C	T1 = (30700, 45000, 45000)	0.9207
	T3 = (9300, 90000, 90000)	0.9207
	T5 = (13600, 135000, 135000)	0.9207

Table 4. Tasksets used in the experiments

Each taskset was executed using different power aware scheduling algorithms. The energy and power consumption for the tasksets are presented in tables 5, 6 and 7. We also present the ratio between energy consumption when using each of the schemes compared to energy consumption when using no scheme at all. For the power consumption measurement we used a National Instruments data acquisition board. We measured power consumed by the processor since we are providing separate power supply for it. For this we used a shunt resistor with very small resistance (0.02 ohm) so that we can measure the voltage drop across it. The voltage drop and the voltage supply value give us the power consumption of the processor. Our measurement setup is similar to the one described in Farkas' work [5] with the difference that we have to sample the supply voltage as well since we are changing it dynamically. We start doing measurements right before the tasks start execution and we stop it when the MPEG application gets done decoding frames. The DAQ board is triggered from software by pulling signals out of the peripheral bus of the 80200 board so that we precisely know where the measurements started and where they ended.

Scheme	Processor			
	Energy (Joules)	Power (Watts)	Ratio	Dead. miss
Normal	39.085	0.779	1	0/0/0
Only Shutdown	31.504	0.628	0.80	0/0/0
Shut./Static	32.024	0.638	0.81	0/0/0
Shut./Static/Dyn.	28.496	0.568	0.72	1/1/2
Shut./Static/Dyn./Adapt. (0.95)	26.581	0.527	0.68	3/2/1
Shut./Static/Dyn./Adapt. (0.90)	26.258	0.522	0.67	3/2/1
Shut./Static/Dyn./Adapt. (0.85)	25.251	0.502	0.64	3/1/4
Shut./Static/Dyn./Adapt. (0.80)	24.835	0.494	0.63	3/2/51
Shut./Static/Dyn./Adapt. (0.75)	24.330	0.483	0.62	3/2/63

Table 5. Energy and Average Power Consumption for Taskset A as shown in Table 4. The total number of executions instances during the measurements for each task of the taskset is 415/207/138.

Scheme	Processor			
	Energy (Joules)	Avg. Power (Watts)	Ratio	Dead. miss
Normal	12.546	0.798	1	0/0/0
Only Shutdown	11.265	0.716	0.89	0/0/0
Shut./Static	9.819	0.624	0.78	1/0/1
Shut./Static/Dyn.	9.811	0.624	0.78	1/0/1
Shut./Static/Dyn./Adapt. (0.95)	9.795	0.623	0.78	1/0/1
Shut./Static/Dyn./Adapt. (0.90)	8.815	0.562	0.70	1/1/31
Shut./Static/Dyn./Adapt. (0.85)	8.828	0.562	0.70	1/1/31
Shut./Static/Dyn./Adapt. (0.80)	8.185	0.522	0.65	34/10/34
Shut./Static/Dyn./Adapt. (0.75)	8.211	0.525	0.65	34/10/34

Table 6. Energy and Average Power Consumption for Taskset B as shown in Table 4. The total number of executions instances during the measurements for each task of the taskset is 130/65/43.

Scheme	Processor			
	Energy (Joules)	Avg. Power (Watts)	Ratio	Dead. miss
Normal	13.080	0.838	1	0/0/0
Only Shutdown	12.342	0.772	0.94	0/0/0
Shut./Static	12.391	0.789	0.94	0/0/0
Shut./Static/Dyn.	10.892	0.693	0.83	0/1/18
Shut./Static/Dyn./Adapt. (0.95)	10.958	0.697	0.83	0/1/18
Shut./Static/Dyn./Adapt. (0.90)	9.875	0.627	0.75	1/8/32
Shut./Static/Dyn./Adapt. (0.85)	9.990	0.637	0.76	11/16/32
Shut./Static/Dyn./Adapt. (0.80)	9.889	0.631	0.75	11/16/32
Shut./Static/Dyn./Adapt. (0.75)	9.789	0.624	0.74	11/16/32

Table 7. Energy and Average Power Consumption for Taskset C as shown in Table 4. The total number of executions instances during the measurements for each task of the taskset is 130/65/43.

In Tables 5, 6 and 7 the scheme column shows which algorithm or combination of algorithms was used. For the adaptive algorithm the number between parentheses represents the values of M. The results presented are expected. Less energy is consumed as long as more agressiveness is introduced to the system. Savings up to 38% are obtained when using the adaptive scheme along with the others. On the other hand, more deadlines are missed with more agressiveness as well. We see that 45% of the deadlines are missed in taskset A when $M = 0.75$. This percentage gets even worse for the other tasksets for less savings. Therefore a proper tradeoff must be found for each different taskset in view of application needs.

6.1. Battery driven example

To complement our goal of showing the utility of PASA we also present a battery-driven application level strategy to enhance battery lifetime of a StrongARM based Compaq PDA called Ipaq. Note that PASA also works on this platform with all the functions presented in Table 1 and also running eCos. We present the results of varying the parameters of a wavelet-based image compression algorithm [17] in order to tradeoff image quality and energy consumption. The first parameter used was BPP or bit per pixel. A higher BPP results in a larger size of the compressed image and higher energy consumption for a better image fidelity. The second parameter used is the decomposition level (L). The larger the L the better is the quality of the image and also the more energy it consumes.

The image compression algorithm is ran in a continuous loop with battery polling (using the function `pahal_battery_get_info()`) being performed every 10 seconds. On the top of this a simple power tradeoff policy is added to adapt the quality of the image against the battery voltage left. The net effect of this policy is that whenever the battery drops 30mV the application adjusts the image BPP by -0.5 starting at 1.5. Figure 4 shows the result. At the beginning voltage drops at the speed of $BPP = 1.5$. After a 30mV the application changes BPP to 1.0 and the voltage drop slows down. After a new 30mV drop the application changes BPP to 1.0. For a cut-off of 4020mV the battery lasts for 340 seconds, as opposed to 290 seconds, when adapting the application to the voltage left in the battery. Therefore the battery life is extended by 18% with a slight degradation of image quality.

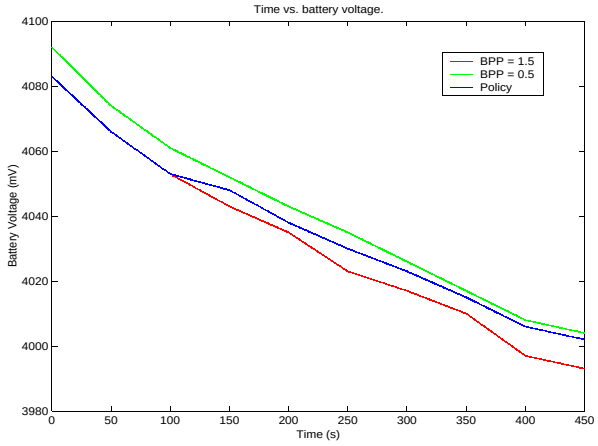

Figure 4. Battery lifetime when adapting the image compression application to the remaining battery left

7. CONCLUSIONS

In this paper, we have presented a sofware architecture, PASA, that enables communication of energy related data among applications,

RTOS and hardware. In order to show its utility, PASA was incorporated into *eCos* operating system running on a complete variable voltage system based on a XScale processor. We implemented four different DVS algorithms, from a simple shutdown based scheme to a dynamic predictive and adaptive DVS algorithm and collected data to show the energy savings on a real system implemented using PASA. The results show a gain of up to 38% when compared to the execution without any power management incorporated. We also presented a battery-driven application adaptation scheme using PASA. For future we will add more power aware services to the RTOS such as memory management and I/O and augment PASA with functions to access these services. We are also investigating ways through which the OS can direct application adaptation in order to achieve a better tradeoff performance / power.

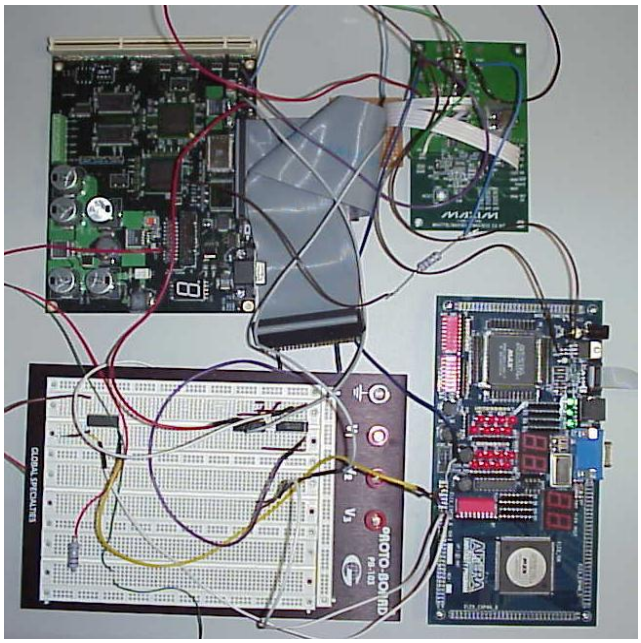

Figure 5. Picture of the hardware used. The Xscale board is in the left upper side. The Maxim Board in the right upper side. In the left lower side one can see a bread board with some interconnections such as signals to trigger the DAQ board. Finally at the right lower side one can see the FPGA board, responsible for generating interrupts.

8. REFERENCES

[1] A. Acquaviva, L. Benini, and B. Riccó. Energy characterization of embedded real-time operating systems. In *Proceedings of the Second Workshop on Compilers and Operating System for Low Power (COLP'01)*, pages 5–1–5–7, Barcelona, Spain, September 2001.

[2] L. Benini, A. Bogliolo, and G. de Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, June 2000.

[3] A. P. Chandrakasan and R. W. Brodersen. *Low Power CMOS Digital Design*. Kluwer Academic Publishers, Norwell, MA, 1996.

[4] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Proceedings of the Design Automation Conference*, pages 312–315, June 2000.

[5] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J.-A. M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Measurement and Modeling of Computer Systems*, pages 252–263, 2000.

[6] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, December 1999.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Mobile Computing and Networking*, pages 13–25, 1995.

[8] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *International Symposium on Low Power Eletronics and Design*, Huntington Beach, CA, USA, 2001.

[9] I. Hong, M. Potkonjak, and M. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processors. In *Proceedings of the International Conference on Computer Aided Design*, 1998.

[10] Intel. Intel 80200 processor evaluation platform board manual (80200EVB), 2001.

[11] Intel and Microsoft. Bios interface specification, 1996.

[12] Intel, Microsoft, and Toshiba. Advanced configuration and power interface specification, 1996.

[13] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processor. In *International Symposium on Low Power Eletronics and Design*, pages 197–202, August 1998.

[14] C. M. Krishna and Y. H. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington D.C, May 2000.

[15] P. Kumar and M. Srivastava. Predictive strategies for low-power rtos scheduling. In *Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 343–348, 2000.

[16] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Battery-driven system design: A new frontier in low power design. In *Asia and South Pacific Deisgn Automation Conference (ASP-DAC) / International Conference on VLSI Design*, January 2002. Invited Embedded Tutorial.

[17] J. Lehtinen. Gwic - gnu wavelet image codec. in http://jole.fi/research/gwic/.

[18] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[19] J. W. S. Liu. *Real Time Systems*. Prentice Hall, 2000.

[20] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *In Proceedings of 18th Symposium on Operating Systems Principles*, 2001.

[21] RedHat. eCos: Embedded configurable operating system. http://www.redhat.com/products/ecos/, 1998.

[22] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the Design Automation Conference*, pages 134–139, 1999.

[23] M. Srivastava, A. P. Chadrakasan, and R. Broderson. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on VLSI Systems*, 4(1):42–55, 1996.

[24] A. Vahdat, A. R. Lebeck, and C. S. Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[25] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, California, U.S., 1994.